Adaptive Dynamic Load Balancing in Heterogenous Multiple GPUs-CPUs Distributed Setting: Case Study of B&B Tree Search

Trong-Tuan Vu, Bilel Derbel, Nouredine Melab

Adaptive Dynamic Load Balancing in Heterogeneous Multiple GPUs-CPUs Distributed Setting: Case Study of B&B Tree Search

Trong-Tuan Vu, Bilel Derbel, and Nouredine Melab

DOLPHIN, INRIA Lille - Nord Europe, University Lille 1, France
firstname.lastname@inria.fr

Abstract. The emergence of new hybrid and heterogenous multi-GPUs multi-CPUs large scale platforms offers new opportunities and poses new challenges when solving difficult optimization problems. This paper targets irregular tree search algorithms in which workload is unpredictable. We propose an adaptive distributed approach allowing to distribute the load dynamically at runtime while taking into account the computing abilities of either GPUs or CPUs. Using Branch-and-Bound and FlowShop as a case study, we deployed our approach using up to 20 GPUs and 128 CPUs. Through extensive experiments in different system configurations, we report near optimal speedups, thus providing new insights into how to take full advantage of both GPUs and CPUs power in modern computing platforms.

1 Introduction

Context and Motivation. The current trend in high performance computing is converging towards the development of new software tools which can be efficiently deployed over large scale hybrid platforms, interconnecting several hundreds to thousands of *heterogeneous* processing units (PUs) ranging from multiple distributed CPUs, multiple shared-memory cores, to multiple GPUs. Although the aggregation of those resources can in theory offer an impressive computing power, achieving high performance and scalability is still bound to the expertise of programmers in developing new parallel techniques and paradigms operating both at the algorithmic and at the system levels. The heterogeneity and incompatibility of resources in terms of computing power and programming models, make it difficult to parallelize a given application without significantly drifting away from the optimal and theoretically attainable performance. In particular, when parallelizing highly irregular applications producing unpredictable workload *at runtime*, mapping dynamically generated tasks into the hardware so that workload is distributed evenly is a challenging issue. In this context, adjusting the workload distributively is mandatory to maximize resource utilization and to optimize work balance over massively parallel and large scale distributed PUs.

In the optimization field, irregular applications producing dynamic workload do not stand for an exception. Many search algorithms operating in some decision space are essentially dynamic and irregular, meaning that neither the search trajectory nor the amount of work can be predicted in advance. For instance, while some search regions

may require much computational efforts to be processed, some others may require only a few. This is typically the case of several tree search algorithms coming from discrete and combinatorial optimizations, artificial intelligence, expert systems, etc. Generally speaking, this paper is targeting tree search-like algorithms endowed with some splitting/selection, pruning/elimination and evaluation/bounding strategies to decide on what to explore/search next. Despite the possibly sophisticated and efficient strategies one can design, these kinds of algorithms still undergo a huge amount of processing time when tackling large scale and/or difficult problems. More importantly, the knowledge acquired during the search changes dynamically the shape of the tree. Hence, it ends up with an unpredictable search process producing a highly variable amount of work. From parallel computing and high performance perspectives, these algorithms can be viewed as 'skillful' adversaries which are very difficult to counteract efficiently.

The goal of this paper is to push forward the design of parallel and distributed optimization algorithms requiring dynamic load balancing, in order to run them efficiently on heterogenous systems consisting of multiple CPUs coupled with multiple GPUs. More precisely, we consider the case study of the Branch-and-Bound (B&B), viewed as a generic algorithm searching in a dynamic tree representing a set of candidate solutions built dynamically at runtime. Given that several distributed CPUs and GPUs coming from possibly different clusters connected through a network can be used to parallelize the B&B tree search, three major issues are addressed:

Q1. Can we benefit from the different degrees of parallelism available in the tree search procedure and map them efficiently into the different PUs?

Q2. Given no knowledge about the amount of work the search would produce, can we distributively coordinate PUs so that parallelism dynamically unfolds, while communication cost and idle time of PUs are kept minimal?

Q3. Having PUs with different computing abilities, can we distribute the load evenly in order to attain optimal speedup while scaling the network?

Contribution overview. In this paper, we answer the three previous questions in the positive while giving new insights into how to fully benefit from heterogenous computing systems and solve difficult optimization problems. More precisely, we describe a two-level and fully distributed parallel approach taking into account PU characteristics. Our approach incorporates an adaptive dynamic load balancing scheme based on distributed work stealing, in order to flow workloads efficiently from overloaded PUs to idle ones at runtime. Furthermore, it does not require any parameter tuning or specific optimization operations so that it is adaptive to heterogeneous computing systems. We implemented and deployed our approach over a distributed system of up to 20 GPUs and 128 CPUs coming from three clusters. Different scales and configurations of PUs were experimented with the B&B algorithm and the well-known FlowShop combinatorial optimization problem [14] as a case study. Firstly, on one single GPU, we improve on the running time of the previous B&B GPUs implementations [4, 11] by at least a factor of two on the considered instances (the speedup with respect to one CPU is around $\times 70$). More importantly, independently of CPUs or GPUs scale or power, our approach provides a substantial speed-up which is *nearly optimal* compared to the ideal performance one could expect in theory. It is worth to notice that although our experimentations are conducted for the specific FlowShop problem, it is generic in the sense

that it undergoes no specific optimization with respect to neither B&B nor FlowShop. Therefore, it can be appropriate to solve other optimization problems, *as far as a GPU parallel evaluation (bounding) of search nodes (viewed as a blackbox)* is available.

From the optimization perspective, relatively few investigations are known on heterogenous parallel tree search algorithms. Specific to B&B, some very recent GPU parallelizations are known for some specific problems [3, 4, 11, 2, 10]. The focus there is mainly on the parallelization of the bounding step which is known to very time-consuming. The only study we found on aggregating the power of multiple GPUs presents a Master/Slave-like model and an experimental scale of 2 GPUs [4]. The authors there stressed more on the parallel design issues and not on scalability nor performance optimality. They reported a good but sub-optimal speed-up when using 2 GPUs, which witness the difficulty of the problem. To the best of our knowledge, the new parallel approach presented in this paper is not only the first to scale near linearly up to 20 GPUs but also the first to address the joint use of multiple distributed CPUs in the system.

From the parallel perspective, very few works exist on the parallelization of highly irregular applications in heterogenous platforms. In particular, we found no in-depth and systematic studies of application speed-up at different CPU-GPU scales. Knowing that the adaptive workload distribution strategy adopted in this paper is generic and not specific to tree search or B&B, our study provides new insights into the scalability of distributed protocols harnessing *both* multiple GPUs and CPUs which have a substantial gap in their respective computing power.

Outline. In Section 2, we draw the main components underlying our distributed approach while motivating their design architecture. A more detailed and technical description then follows in Section 3. In Section 4, we report and discuss our experimental results. In Section 5, we conclude the paper and raise some open issues.

2 A comprehensive overview of our approach

In this section, we give the general design principles guiding our approach. The goal is to introduce different components of our approach in a comprehensive manner without going into system technicalities or implementation details.

2.1 Application Model and Preliminaries

To simplify the presentation and clarify our contribution, let us model the B&B algorithm, as a tree search algorithm that starts from a root node representing an optimization problem. During the search, a parent node generates new child nodes (e.g., representing partial/complete candidate solutions) at runtime. The quality of these nodes is evaluated (bounding) using a given (heuristic) procedure. Then, according to the search state, some nodes are discarded (pruning) whether some others can be selected and the tree is expanded (branching) to push the search forward and so on. Having this in mind, the general architecture of our approach for distributing search computations is depicted in Fig. 1 and discussed in the following subsections. Each subsection will give an answer to one of the three questions addressed in the introduction.

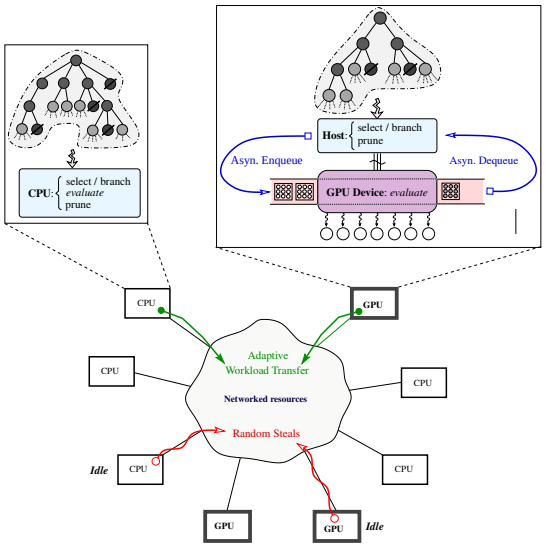

Fig. 1: Overview of our parallel approach

2.2 A two-level parallelism (Q1)

As shown in Fig. 1, our approach is based on two levels of parallelism mapping the search into possibly multiple CPUs and multiple GPUs. In *Level 1*, different CPUs or GPUs can explore different subtrees in parallel, i.e., select, branch, evaluate, prune, etc. As it will be discussed later, enabling the distribution of subtrees over PUs dynamically at runtime is at the heart of our approach. In *Level 2*, the evaluation of tree nodes (bounding for B&B) is done *inside* every GPU device, while the other search operations are performed *in parallel* by the GPU host, i.e. CPU. In fact, due to the irregularity and unpredictable shape of the tree, it is well understood that implementing the whole search operations inside GPU, could suffer from the thread divergence induced by the SIMD programming model of GPUs. The evaluation step, on the other side, can highly benefit from the parallelism offered by the many GPU cores. These design/model choices are essentially motivated by the fact that the evaluation phase of many combinatorial optimization problems is very time-consuming, e.g bounding for B&B, so that it dominates the other operations.

Although the GPU device can handle the evaluation of many tree nodes in parallel [4, 10], the CPU host still has to prepare a data containing these nodes, copy them into GPU memory and copy back the result. This implies that while computations are carried out on the GPU device, the host is idle and vice-versa. In our Level 2 parallelism, the host and the device are managed to run computations *in parallel*, i.e., while the device is evaluating tree nodes, the host is preparing new data for the next evaluation in the device. Notice that the evaluation step of many tree nodes inside the GPU is of course implying another type of parallelism which we will not address in this paper, since our focus is on scalability and work distribution on multiple PUs.

2.3 Dynamic work stealing (Q2)

It is essential to fully explore the computing resources provided by a single CPU-GPU. However, it is more challenging to fully utilize the networked resources available in the distributed system. In fact, the irregularity generated at runtime can eventually lead to very poor performances because most computing nodes are underloaded and few others are highly overloaded, or because of the cost of synchronizing PUs and transferring work is so high. In this paper, we propose a distributed work stealing [7] based mechanism to tackle this issue. If a PU runs out of work, it acts like a thief and tries to steal work (i.e., subtree nodes) from another PU, called victim, *chosen uniformly at random*. This simple decentralized work stealing approach is motivated by two facts. Firstly, idle PUs acquire work cooperatively in parallel, thus eliminating the time required to synchronize, and to transfer/distribute data among them. In particular, no PU can constitute a communication bottleneck, so that the protocol would not suffer from scalability issues. Secondly, random work stealing (RWS) in shared memory is theoretically shown to give good performance under some application circumstances [1]. However, it has not been studied so far in a heterogenous networked setting involving the cooperation of both CPUs and GPUs at large scales in order to solve hard optimization problems.

2.4 Adaptive work balancing (Q3)

One crucial issue in RWS for efficient dynamic load balancing is the amount of work, denoted f, to be transferred between thieves and victims. Generally, the thief attempts to balance the load evenly between itself and the victim. In fact, when this amount of work is very small, the large overhead is observed since many load balancing operations are performed. At the opposite, when it is very large, too few load balancing operations will occur, thereby resulting in large idle times despite the fact that surplus work could be available. In classical RWS approaches, this is a hand-tuned parameter which depends on the distributed system and the application context [12]. In a theoretical study [1], the stability and optimality of RWS can be analytically guaranteed for $f \leq 1/2$. In practice, the so called steal-half strategy ($f = 1/2$) is often shown to perform efficiently using homogenous computing units. In a heterogenous and large scale scenario, this parameter is even more sensitive because of the wide variety of computing capabilities of different PUs. In this context, the community lacks relatively much knowledge to understand how to attain good performance for RWS based protocols.

To understand the issues we are facing when distributing tree search works over multiple CPUs and GPUs, one has to keep in mind that (i) a GPU is substantially faster in evaluating tree nodes than a CPU, (ii) nothing can be assumed about the amount of tree nodes initially. Hence, if GPUs run out of work and stay idle searching for work, the performance of the system can drop dramatically. If only few CPUs are available in the system, work stealing operations from CPUs to GPUs can cause a severe penalty to performance. This is because the few CPUs can only contribute very little to the overall performance but their stealing operations to GPUs can disturb the GPU computations and prevent them from reaching their maximal speed. In contrast, if work is scheduled more on GPUs, then a significant loss in performance can occur when a relatively large number of CPUs are available. To tackle these issues, we propose to configure RWS so

that when performing a steal operation, the value of f is computed at runtime based on the normalized power of the thief and the victim, where the computing power of every PU is estimated continuously at runtime with respect to the application being tackled.

3 Parallel and Distributed Protocol Details

3.1 Concurrent computations for single CPU-GPU (*Level 2* parallelism)

Generally speaking, an application is composed of multiple tasks and each task could be executed on a GPU or CPU depending on its characteristics. For each task running on a GPU, input data is transferred to GPU memory, a kernel is executed on the input and the outputs are copied back to the host for being processed. In other words, standard CPU/host GPU/device executions are synchronized sequentially. While the host is working, i.e to prepare/process input/output data, the device sits idle, and vice versa. This can significantly slow down computations especially when the host and the device can perform concurrent operations *in parallel*.

With the rapid evolving of GPU devices, it is now possible to address the above issue by carefully exploiting the new available hardware and software technologies. For instance, NVIDIA GPUs with compute capability ≥ 1.1 are associated with a *compute* engine and a *copy* engine (DMA engine). NVIDIA's Fermi GPUs have up to 2 copy engines, one for uploading from host to device and one for downloading from device to host. Each engine is equipped with a queue to store pending data and kernels that will be processed by the engine shortly.

The *Level 2* host-device parallelism discussed in our approach can be enabled using CUDA primitives as sketched in Algorithm 1. Each ENQUEUE procedure dispatches CUDA operations into the GPU device *asynchronously*, i.e pushes/retrieves data and launches the kernel. This is possible by wrapping those operations into a CUDA stream. All operations inside the same CUDA stream get automatically synchronized and executed sequentially, but the CUDA operations of different streams could overlap one with the other, e.g., execute the kernel of stream 1 and retrieve data from stream 2 concurrently in parallel. In our implementation, we use a maximum number of streams, i.e., variable $r_{\max}$, which is the maximum number of elements *(data, kernel)* in the queue of GPU Copy engine and Compute engine. The maximum number of streams that a GPU can handle depends in general on GPU global memory characteristics. For B&B search, data is a pool of tree nodes and kernel is the bounding function. Asynchronously in parallel to the ENQUEUE procedure, the DEQUEUE procedure in Algorithm 1 waits for data copied back from the device on a given CUDA stream, and processes the output data. In our B&B implementation, this corresponds to the pruning operation. Notice also that Algorithm 1 is independent of the specific data or kernels being used, so that it can be customized with respect to the search operations or optimization problems at hand. In particular, any existing kernel implementing parallel tree node evaluation is applicable.

3.2 Distributed Work Stealing for multiple CPUs/GPUs (*Level 1* parallelism)

In this section, we describe how our *Level 1* parallelism is implemented, i.e., how tree nodes are distributed over PUs. As discussed previously, this is based on an adaptive

Algorithm 1: GPU *Level 2* parallelism — Concurrent host-device template

Data: q_host, q_device: queue of $task$ in host and GPU; q_host_size: current size of q_host (0 initially); $stream[r_{\max}]$: CUDA Stream of $r_{\max}$ elements; w_index, r_index: next index to write (resp. read) to (resp. from) the queues (0 initially).

1 **while** *tree nodes are available* **do in parallel:**

```
// Push tree nodes for evaluation inside GPU
```

2 **Execute Procedure** ENQUEUE;

```
// Retreive and process evaluated nodes from the GPU
```

3 **Execute Procedure** DEQUEUE;

Procedure Enqueue

1 **while** $q\_host\_size < r_{\max}$ **do**

2 $q\_host[w\_index].task \leftarrow$ prepare a pool of tree nodes;

```
// Asynchronous Operations on stream[w_index]
```

cudaMemcpyAsyn($q\_device[w\_index]$, $q\_host[w\_index]$, $sizeof$(q_host[w_index].task),

3 ***cudaMemcpyHostToDevice, stream[w_index]***);

```
// Launch parallel evaluation (bounding) on device
```

KERNEL<<< ***stream[w_index]*** >>> ($q\_device[w\_index]$) ;

4 cudaMemcpyAsyn($q\_host[w\_index].bound$, $q\_device[w\_index].bound$,

5 $sizeof(q\_device[w\_index].bound)$,

6 ***cudaMemcpyDeviceToHost, stream[w_index]***) ;

7 $w\_index \leftarrow (w\_index + 1) \pmod{r_{\max}}$; $q\_host\_size \leftarrow q\_host\_size + 1$;

Procedure Dequeue

1 **if** $q\_host\_size > 0$ **then**

```
// Wait for results from device on stream[r_index]
```

cudaStreamSynchronize(***stream[r_index]***) ;

2 Process output data from $q\_host[r\_index]$, i.e., prune nodes ;

3 $r\_index \leftarrow (r\_index + 1) \pmod{r_{\max}}$; $q\_host\_size \leftarrow q\_host\_size - 1$;

work stealing paradigm and sketched in Algorithm 2 below — Notice that Algorithm 2 is to be executed *distributively by each* PU, i.e., v variable.

Algorithm 2: *Level 1* Parallelism — Distributed Adaptive Work Stealing

1 **while** *Termination not detected* **do in parallel:**

2 **Execute Procedure** THIEF;

3 **Execute Procedure** VICTIM;

Stealing Granularity. To efficiently balance the work load, stealing granularity, that is the amount of work to be transferred from victims to thieves, plays a crucial role. Depending on the hardware platform and the input application, there may exist a value of work granularity giving the best performance. For instance, for the Unbalanced Tree

Procedure Thief

```
1 x ← runtime normalized computing power ;
2 repeat
3     u ← pick one PU victim uniformly at random ;
        // v denotes the actual thief PU executing the procedure
4     Send a steal request message (v, x) to u;
5     Receive u's response (reject or work) message ;
6 until some tree nodes are successfully transferred from victim u;
```

Procedure Victim

```
1 if a steal request is pending then
2     y ← runtime normalized computing power ;
3     if tree nodes are available then
4         (v, x) ← pull the next pending thief request;
5         work ← share tree nodes in the proportion of x/(x+y) ;
6         Send back shared work to v ;
7     else
8         Send back a reject message to v ;
```

Search benchmark [6], which is often considered as an adversary application to load balancing [9, 13], it was shown that steal half works best for binomial trees. Instead, stealing a fixed amount of work items (i.e., 7 items) is shown to work well for geometric trees. Besides, in a heterogeneous and hybrid computing system, the hardware characteristics of PUs, e.g., clock speed, Cache, RAM, etc, can be highly needed to balance the work load evenly depending on the characteristics of every available PU. Because high variations in computing power among PUs can lead to high imbalance and idle times, one has also to manage this issue carefully when distributing work. One possible solution to the above issues could be to profile the system components/PUs and tune work granularity offline before application execution in order to get the best performance. It should be clear that such an approach is not reasonable nor feasible, for instance when the system may undergo a huge number of many different types of PUs, or when having many different applications at hand.

In our stealing approach, we make every PU maintain at runtime a measure reflecting its computing power, i.e., variable x in Algorithm 2. As the computations are running on, every PU adjusts its measure continuously with respect to the work processed in the previous iterations. In our approach, we simply use the average time needed for processing one tree node. More precisely, each PU sets its computing power to be $x = N/T$, where T is the (normalized) time elapsed since the PU has started the computation and N is the number of tree nodes explored locally by that PU. Notice that time T includes, in addition to tree node evaluation (i.e. B&B lower bounding), the time needed for other search operations (i.e. select, branch and prune) but *not* the time when a PU stays idle. When running out of work, a PU v then attempts to steal work

by sending a request message to one other PU u chosen at random, while wrapping the value of x in the request. If a victim has some work to serve, then the amount of work (i.e., number of tree nodes) to be transferred is in the proportion of $x/(x+y)$, where y is the computing power maintained locally by the victim. Otherwise, a reject message is sent back to notify the thief and a new stealing round is performed. Initially, the value of x is normalized so that all PUs have the same computing ability. In other words, the system starts stealing half and then the stealing granularity is refined for each pairwise PU. Intuitively, each PU acts as a black-hole, so that the higher computing power of PUs is, the more available work are flowed to the black-hole. Furthermore, no knowledge about PUs is needed so that any performance variation at system/application level would also be detected at runtime.

Termination Detection. One issue in the template of Algorithm 2 is how to decide on termination distributively (Line 1). For B&B, this occurs when all tree nodes are explored (explicitly or implicitly, i.e., pruned). However, since stealing is performed locally by idle PUs, the work remaining in the system is not maintained anywhere. This is a well understood issue for which an abundant literature can be found [5].

We use a fully distributed scheme, in which PUs are mapped into a tree overlay and the termination is detected in an 'Up-Down' distributed fashion. In the up phase, if a PU becomes idle and has not served any stealing request, it will then integrate a positive termination signal to its children signals. If a PU turns to idle and has served at least one stealing request, it will then integrate a negative termination signal to its children signals. Then the termination signal is forwarded to the parent and eventually to the root. In the down phase, if the root receives at least one negative termination signal from its children, it broadcasts a signal to restart a new round of termination detection. Otherwise, if only positive termination signals are received, the root broadcasts a message to announce global termination. The tree overlay used in our implementation is a binary one so that PU degrees and the overlay diameter are kept low. This allows us to scale out PUs while avoiding communication bottlenecks and performance degradation once a termination phase is performed.

Knowledge Exchange. An important ingredient missing to complete our approach, is the mechanism allowing PUs to exchange knowledge during the search. In B&B for instance, one important issue is to share the best upper bound found by any PU in order to avoid exploring unnecessary branches. We use the same tree overlay topology used in the above scheme for termination detection, to propagate search knowledge (new upper bounds) among PUs. Since the overlay diameter is logarithmic, propagating knowledge among PUs has a relatively limited communication cost.

4 Experimental Results

4.1 Experimental setting

We consider the permutational FlowShop problem with the objective of minimizing the makespan (C_{max}) of scheduling n jobs over m machines as a case study in our experiments. The well-known Taillard' instances [14] of the family of 20 jobs and 20

machines are considered. To give an idea of the their difficulties, the time required for solving these instances on a standard modern CPU, starting from scratch (that is without any initial solution), can be several dozens of hours.

Our approach needs three major components to be experimented: (i) the distributed load balancing protocols (*Level 1* parallelism), (ii) the concurrent host-device computations (*Level 2* parallelism) and (iii) the GPU kernel for bounding w.r.t FlowShop. The GPU kernel was taken to be the one of [4, 11] and used as a blackbox. *Level 1* (resp. *2*) was implemented using low level c++ libraries (resp. c++ concurrent threads and CUDA primitives). Three clusters C_1, C_2 and C_3 of the Grid'5000 French national platform [8] were involved in our experiments. Cluster C_1 contains 10 nodes, each equipped with 2 CPUs of 2.26Ghz Intel Xeon processors with 4 cores per CPU. Besides, each node is coupled with two Tesla T10 GPUs. Each GPU contains 240 CUDA cores, a 4GB global memory, a 16.38 KB shared memory and a warp size of 32 threads. Cluster C_2 (resp. C_3) were equipped with 72 nodes (resp. 34 nodes), each one equipped with 2 CPUs of 2.27 Ghz Intel Xeon processor with 4 cores per CPU (resp. 2 CPUs of 2.5 Ghz Intel Xeon processor having 4 cores) and a network card Infiniband-40G.

Let us point out that the GPU *kernel* implementation of [4, 11] has a parameter s referring to the maximum number of B&B tree nodes that are pushed into GPU memory for parallel evaluation. It is shown in [11] that the parameter s has to be fixed to a value $s^\star$ so that the device memory is optimized and the performance is the best on a single GPU. In [4], it is shown how to tune the value of s online so that it converges to $s^\star$. Since we assume that the GPU kernel is provided as a black-box, and unless stated explicitly, the value of s is fixed in our experiments to be simply $s^\star$. In our experimental study, we are also interested in analyzing how our approach would perform when having GPU kernels allowing for different speed-ups in the evaluation phase. This can be typically the case for other type of problems, different hardware configurations, etc. Being able to understand whether our load balancing mechanism is efficient in such a heterogenous setting, independently of the considered scale or speedup gap between available CPUs and GPUs, is of great importance. In this paper, we additionally view the parameter s as allowing us to empirically reduce the intrinsic speed of a single GPU, and thus to experiment our approach while using different GPU and CPU configurations. In the remainder, we shall consider the following scenarios:

1. Enabling our *Level 2* parallelism within a single CPU-GPU.
2. Running our approach (*Level 1* + *Level 2*) at different scales with multiple GPUs.
3. Running our approach with a fixed number of GPUs, while scaling the CPUs.
4. Running our approach with a fixed number of CPUs, while scaling the GPUs.
5. Running our approach with CPUs and GPUs having different computational powers.

In all scenarios, a GPU device is launched with 1 CPU core taken from C_1. For the first four scenarios, CPUs are taken from cluster C_2. As for the fifth scenario, we mix CPUs of different hardware clock speeds, taken from C_2 and C_3, and GPUs launching kernels configured with different values of s. The previous scenarios aim at providing insights on how the system performs independently of the scale and/or the power of CPUs and GPUs. For all experiments, we measure T and N, respectively the time needed to complete the B&B tree search and the number of B&B tree nodes that were

effectively explored. All reported speedups are relative to the number of B&B tree nodes explored by time units, that is N/T.

4.2 Impact of asynchronous data transfer on a single GPU

We start our analysis by evaluating the impact of *Level 2* host device concurrent operations. For the ten instances in Taillard' family 20*20, we report in Fig 2 execution time and speedup w.r.t. the baseline sequential host-device execution [11], for different number of concurrent CUDA streams (variable $r_{\max}$ in Algorithm 1) and different GPU kernel parameters s. One can clearly see that substantial improvements are obtained, i.e., our approach is at least 2 times faster. It also appears that the maximum number of concurrent CUDA streams $r_{\max}$, which is the only parameter used in our approach, has only a marginal impact on performance. Fig 2 Right shows that the speed-up, w.r.t the sequential host-device execution, is substantial ($> 30\%$) but depends on kernel parameter s. This is because for lower values of s, the host spends more time pushing small amount of data, while the device is less efficient. In other words, *Level 2* parallelism performs better when the amount of data and computations on device is higher.

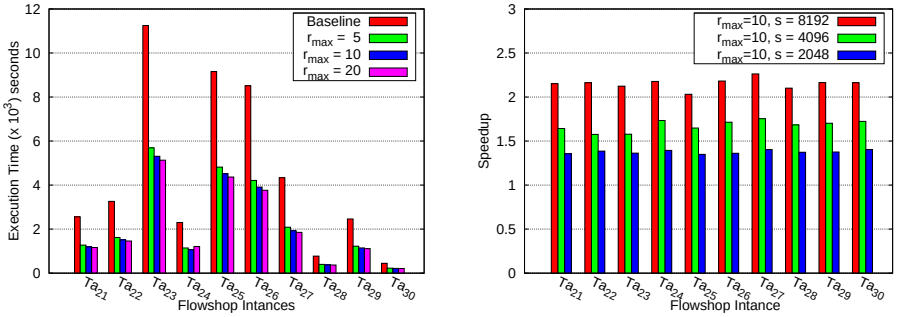

Fig. 2: *Level 2* parallelism *vs.* baseline sequential host-device execution [4]. **Left**: Execution time with different number $r_{\max}$ of CUDA streams and $s = s^{\star}$ (Lower is better). **Right**: Speedup w.r.t baseline for different values of s and $r_{\max} = 10$ (Higher is better).

4.3 Scalability and Stealing Granularity for multiple GPUs

In this section, we study the scalability of our approach when only multiple GPUs are available in the system. For this set of experiments we choose the first instance Ta_{21} to be our case study. In Fig 3 Left, we report the speedup of our approach w.r.t one single GPU, and also the speedup obtained when using a static stealing granularity (with of course *Level 2* parallelism enabled). By static stealing, we mean that we initially fix the proportion of tree nodes to be stolen as a parameter $f \in \{1/2, 1/4, 1/8\}$. Two observations can be made. Firstly, our adaptive approach performs similar to the best static stealing, which is for $f = 1/2$ from our experiments. Other values of f in static stealing are in fact worse especially in high scales. Secondly, we are able to scale linearly

with the number of GPUs. At scale 16, one can notice a slight decrease in speedup. We attribute this to two factors: (i) the communication cost of distributing work strategy to be not negligible in large scales, and (ii) sharable work becomes very fine grain so that it limits the maximal performance of GPUs. Actually, the results of Fig. 3 Left are obtained with parameter s being $s^\star$ the maximal (and best) amount of tree nodes that a single GPU can handle. In Fig 3 Right, we push our experiments further by taking other values for parameter s. We can clearly see that the speed-up (w.r.t. one single GPU running a kernel with the same value of s) is not impacted. The scalability is even slightly better when the kernels are less efficient. This can be interpreted as the scalability of our approach being not sensitive to other system/application settings with GPUs having possibly different processing powers.

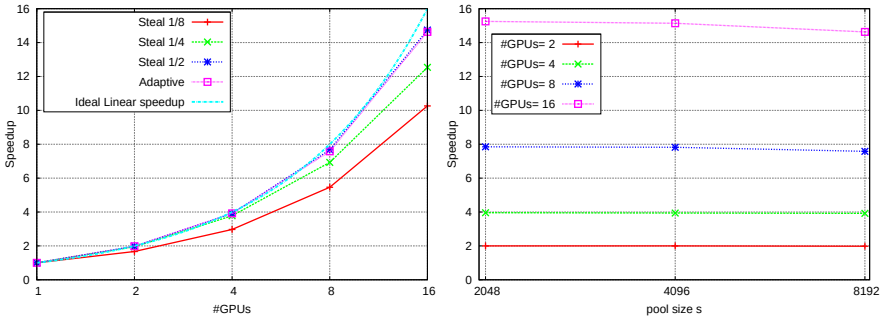

Fig. 3: **Left**: Scalability of our adaptive approach *vs.* static stealing ($s = s^\star$). X-axis refers to the number of GPUs in log scale. Y-axis refers to the speed-up with respect to one GPU. **Right**: Speedups of our approach as a function of s. $r_{\max} = 10$.

4.4 Adaptive Stealing for multiple GPUs multiple CPUs

In this section, we study the properties of our approach when mixing both CPUs and GPUs. For that purpose, we proceed as following. Let α_i^j be the speedup obtained by a *single* PU j with respect to PU i. We naturally define the linear (ideal) normalized speedup *with respect to PU* i, to be $\sum_j \alpha_i^j$. For instance, having p *identical* GPUs and q *identical* CPUs, each GPU being β times faster than each CPU, our definition gives a linear speedup with respect to *one GPU* (resp. *one CPU*) of $p + q/\beta$ (resp. $q + \beta \cdot p$). The following sets of experiments shall allow us to appreciate the performance of our approach when varying substantially the ratio between the number of GPUs and CPUs.

CPU Scaling. In this set of experiments, we fix the number of GPUs and scale the number of CPUs. Besides, we experiment two other static baseline strategies. The first one is the standard steal half strategy. The second one, we term 'Weighted Steal', is hand tuned as following. After profiling the different PUs in the system and running the B&B tree search with the corresponding FlowShop instance on *every single PU until termination*, we provide each PU with the relative computing power of every other PU in the system. Then, the amount of work transferred from PU i to PU j is initially fixed to be in the proportion of the relative computing power observed in the profiling phase.

The results with 1 and 2 (identical) GPUs and (identical) CPUs ranging from 1 to 128 are reported in Fig 4.

One can clearly see that the adaptive approach scales near linearly. It also performs similar to the weighted static strategy while avoiding any tedious profiling and/or PU code configurations. In particular, the weighted strategy cannot be reasonable in production systems with different PU configurations since it requires much time to tune the systems. Turning to the steal half static strategy, it appears to perform substantially worse. When having relatively few CPUs, the performance of steal half is even worse than in a scenario where only GPUs are available (see Fig. 3). It is also getting worse as we push additional few CPUs in the system. Improvements over 1 or 2 GPUs are only observed when the number of CPUs is relatively high (w.r.t GPU power).

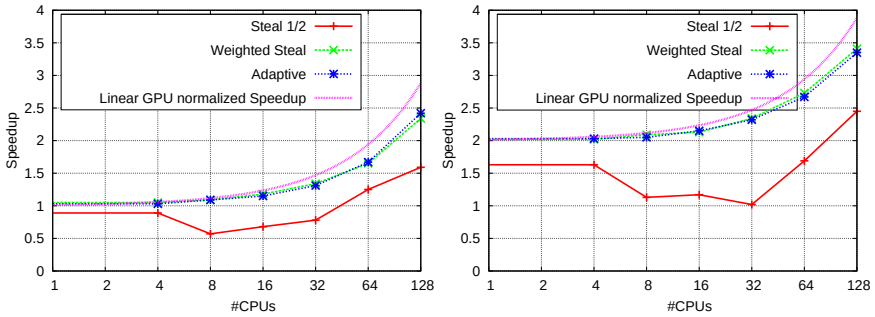

Fig. 4: Speedup of our approach *vs.* static steal when scaling CPUs and using 1 GPU (**Left**) and 2 GPUs (**Right**). X-axis is in the log scale. Speed-up are w.r.t. one GPU. $r_{\max} = 10$.

GPU Scaling. We now fix the number of CPUs and study how the behavior of the system when scaling the number of GPUs. Results with 128 (identical) CPUs and (identical) GPUs ranging from 1 to 16 are reported in Fig 5. We can similarly see that our adaptive approach is still scaling in a linear manner while being near optimal. It is also substantially outperforming the static steal half strategy.

Mixed Scaling. Our last set of experiments is more complex since we manage to mix multiple GPUs with empirically different powers and multiple CPUs with different clock speeds. This scenario is in fact intended to reproduce a heterogenous setting where, even PUs in the same family do not have the same computing abilities. In this kind of scenario, where in addition the power of PUs can evolve, e.g., due to system maintenance constraints or hardware renewals/updates, even a weighted hand tuned steal strategy is not plausible nor applicable. In the results of Fig. 6, we fix the number of CPUs to be 128 with half of them taken from cluster C_2 and the other half from cluster C_3 (C_2 and C_3 have different CPU clock speeds as specified previously). For GPUs, we proceed as following. We use a variable number of GPUs in the range $p \in \{1, 4, 8, 12, 16, 20\}$. For $p > 1$, we configure the system so that $1/2$ of GPUs run a kernel with pool size $s^\star$, $1/4$ of them with pool size $s^\star/2$ and the last $1/4$ of them with pool size $s^\star/4$. Once again our approach is able to adapt the load for this complex

heterogenous scenario and to obtain a nearly optimal speedup while outperforming the standard steal half strategy. From the previous set of experiments we can thus conclude that our approach allows us to take full advantage of both GPU and CPU power independently of considered scales, or any hand tuned parameter.

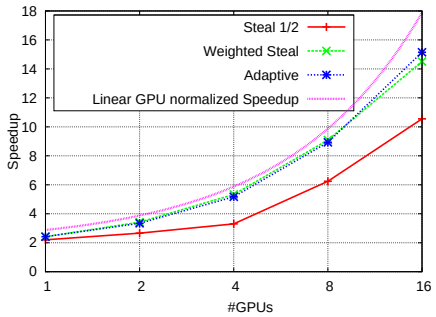

Fig. 5: Speedup (w.r.t. one GPU) when scaling GPUs and using 128 CPUs. $r_{\max} = 10$.

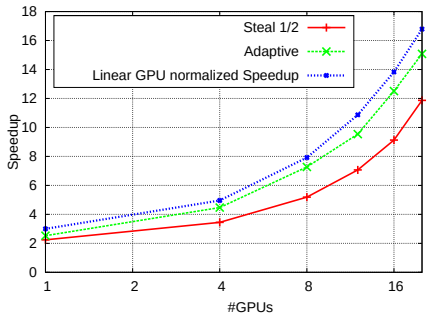

Fig. 6: Speedup when scaling heterogenous GPUs ($1/2$ with $s^\star$, $1/4$ with $s^\star/2$, $1/4$ with $s^\star/4$), and 128 heterogenous CPUs ($1/2$ from cluster C_2, $1/2$ from cluster C_3). Speedup is w.r.t. one GPU configured with $s^\star$. $r_{\max} = 10$.

5 Conclusion

In this paper, we proposed and experimented an adaptive load balancing distributed scheme for parallelizing computing intensive B&B-like tree search algorithms in heterogenous systems, where multiple CPUs and GPUs with possibly different properties are used. Our approach is based on a two-level parallelism allowing for (i) distributed subtree exploration among PUs and (ii) concurrent operations between every single GPU host and device. Through extensive experiments involving different PU configurations, we showed that the scalability of our approach is near optimal, which leaves very little space for further improvements. Besides being able to experiment our approach with other problem-specific GPU kernels, one interesting and challenging research direction would be to extend our approach in a *dynamic* distributed environment where: (i) processing units can join or leave the system, and (ii) different end-users can concurrently request the system for solving different optimization problems. In this setting, the

load has to be balanced not only w.r.t. the irregularity/dynamicity of one single application, but also w.r.t many other factors and constraints that may affect the computing system at runtime.

Acknowledgments

This material is based on work supported by INRIA HEMERA project. Experiments presented in this paper were carried out using the Grid5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`). Thanks also to Imen Chakroun for her precious contributions to the code development of the GPU kernel.

References

1. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, 1999.
2. A. Boukedjar, M.E. Lalami, and D. El-Baz. Parallel branch and bound on a CPU-GPU system. In 20^{th} *Int. Conf. on Parallel, Distributed and Network-Based Processing*, pages 392–398, 2012.
3. T Carneiro, A. E. Muritiba, M. Negreiros, L. De Campos, and G. Augusto. A new parallel schema for branch-and-bound algorithms using GPGPU. In 23^{rd} *Symp. on Computer Architecture and High Performance Computing*, pages 41–47, 2011.
4. I. Chakroun and M. Melab. An adaptative multi-GPU based branch-and-bound. a case study: the flow-shop scheduling problem. In 14^{th} *IEEE Inter. Conf. On High Performance Computing and Communications*, 2012.
5. E. W. Dijkstra. Derivation of a termination detection algorithm for distributed computations. *Control Flow and Data Flow: concepts of distributed programming*, pages 507–512, 1987.
6. J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C-W Tseng. A message passing benchmark for unbalanced applications. *Simulation Modelling Practice and Theory*, 16(9):1177–1189, 2008.
7. Matteo F, Charles E. L, and Keith H. R. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, 1998.
8. Grid500 French national gird. https://www.grid5000.fr/.
9. D. James, L. D. Brian, P. Sadayappan, S. Krishnamoorthy, and N. Jarek. Scalable work stealing. In *ACM Conf. on High Performance Computing Networking, Storage and Analysis*, pages 53:1–53:11, 2009.
10. M. E. Lalami and D. El-Baz. GPU implementation of the branch and bound method for knapsack problems. In *IPDPS Workshops*, pages 1769–1777, 2012.
11. N. Melab, I. Chakroun, M. Mezmaz, and D. Tuyttens. A GPU-accelerated b&b algorithm for the flow-shop scheduling problem. In 14^{th} *IEEE Conf. on Cluster Computing*, 2012.
12. S-J Min, C. Iancu, and K. Yelick. Hierarchical work stealing on manycore clusters. In 5^{th} *Conf. on Partitioned Global Address Space Prog. Models*, 2011.
13. V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In 16^{th} *ACM Symp. on Principles and practice of parallel programming (PPoPP '11)*, pages 201–212, 2011.
14. E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.